Sparstition

A partitioning scheme for large-scale sparse matrix vector multiplication on FPGA

Sigurbergsson, Bjorn; Hogervorst, Tom; Qiu, Tong Dong; Nane, Răzvan

DOI
10.1109/ASAP.2019.00-30

Publication date
2019

Document Version
Final published version

Published in
2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)

Citation (APA)
Sigurbergsson, B., Hogervorst, T., Qiu, T. D., & Nane, R. (2019). Sparstition: A partitioning scheme for large-scale sparse matrix vector multiplication on FPGA. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP): Proceedings* (pp. 51-58). Article 8825125 (2019 IEEE 30TH INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS (ASAP 2019)). IEEE. https://doi.org/10.1109/ASAP.2019.00-30

2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)

Sparstition: A Partitioning Scheme for Large-Scale Sparse Matrix Vector Multiplication on FPGA

Björn Sigurbergsson*, Tom Hogervorst*, Tong Dong Qiu†, Razvan Nane*†
*Delft University of Technology, Delft, The Netherlands
bjornlogi@gmail.com, {t.a.hogervorst, r.nane}@tudelft.nl
†Big Data Accelerate B.V., Delft, The Netherlands
{tong.dong.qiu, razvan.nane}@bigdataccelerate.com

***Abstract*—Sparse Matrix Vector Multiplication (SpMV) is a key kernel in various domains, that is known to be difficult to parallelize efficiently due to the low spatial locality of data. This is problematic for computing large-scale SpMV due to limited cache sizes but also in achieving speedups through parallel execution. To address these issues, we present 1) *sparstition*, a novel partitioning scheme that enables computing SpMV without the need to do any major post-processing steps, and 2) a corresponding HLS-based hardware design that is able to perform large-scale SpMV efficiently. The design is pipelined so the matrix size is limited only by the size of the off-chip memory (DRAM) and not by the available on-chip memory (BRAMs). Our experimental results, performed on a ZedBoard, show that we achieve a computational throughput of up to 300 MFLOPS in single-precision and 108 MFLOPS in double-precision, an improvement of 2.6X on average compared to current state-of-the-art HLS results. Finally, we predict that *sparstition* can boost the computational throughput of HLS-based SpMV kernel to over 10 GFLOPS when using High Bandwidth Memories.**

***Index Terms*—SpMV, SMVM, High-Level Synthesis, FPGA, accelerator, partitioning**

I. Introduction

The Sparse Matrix Vector Multiplication (SpMV) kernel is found in iterative applications across a variety of domains, such as in sparse linear system solvers, PageRank [1], and machine learning [2]. The kernel is typically the bottleneck due to low data locality arising from the random access to either the operand or result vector. Furthermore, there is no reuse of the matrix data which makes the kernel memory-bound and thus dependent on bandwidth for performance.

A lot of effort has been consequently devoted to accelerating SpMV. However, such acceleration attempts faced the additional issue that large-scale matrices do not fit in the caches of the accelerators, motivating the need to perform partitioning. Due to the aforementioned low spatial locality property, the partitioned computation produces intermediate result vectors that need to be added together [3], [4]. This additional post-processing stage is either expensive in terms of hardware or not scalable. Furthermore, it limits the parallelism of the design which may lead to performance degradation. Therefore, it can be desirable to shift some of the complexity to the pre-processing stage and design the hardware accordingly [5].

In order to preserve the parallelism when partitioning is performed and to simultaneously avoid the extra post-processing step, we present *sparstition*. This novel and scalable partitioning algorithm divides the matrix into groups of adjacent rows and compresses each partition via column-shuffling. The resulting partitions are completely disjoint which enables the parallel computation of the result vector. With the *sparstition*, we perform SpMV on matrices that require vectors which are up to 9X larger than the available BRAM storage, and the only limitation is the available DRAM on the ZedBoard. The *sparstition* does incur a pre-processing cost, but we show that this cost is near negligible when it is applied in iterative solvers. Such solvers execute SpMV tens or hundreds of times, or even more frequently, and the *sparstition* typically takes 2-3 iterations before speedup is achieved when using High Bandwidth Memories (HBM) with multiple streams.

Subsequently, a corresponding architecture in Vivado HLS (High-Level Synthesis) is developed to compute partitioned SpMV products efficiently by pipelining memory transfers with the computation. HLS allows for rapid development of hardware architectures but the performance in the literature is currently significantly inferior to custom architectural solutions. We show that by optimizing the bandwidth delivered to the kernel, our kernel design outperforms the state-of-the-art HLS by an average of 2.6X. We also estimate that up to 10 GFLOPS is achievable with our HLS design when using HBMs.

The paper makes the following contributions:

- Sparstition, a novel partitioning algorithm that eliminates the need to create and merge intermediate result vectors. The algorithm works optimally with banded matrices.
- SpMV HLS-based design that maximizes the available bandwidth of the ZYNQ platform and pipelines memory-transfers with the computation. Furthermore, the efficiency of the kernel is evaluated.
- A mathematical model that computes the iteration threshold above which sparstition can be safely applied.
- Experimental results showing a 2.6x speedup over state-of-the-art HLS implementations and a potential 10 GFLOPS when using HBMs.

The rest of the paper is organized as follows: Sec. II gives a brief overview of SpMV and the problems encountered when it is partitioned. It also describes the current state of Vivado HLS and the source of inefficiency with respect to the SpMV

2160-052X/19/$31.00 ©2019 IEEE
DOI 10.1109/ASAP.2019.00-30

kernel. Sec. III summarizes the related work with regards to the existing work for SpMV in HLS and partitioning of sparse matrices. Sec. IV describes *sparstition* using a concrete example. Sec. V details the hardware design used to process the streams generated by sparstition. Sec. VI reports and analyzes the experimental results. Finally, Sec. VII concludes the paper and highlights future work.

II. Background

Many applications in a wide variety of domains across, among others, physics, mathematics, and computer science, use matrices that contain many more zeroes than non-zero values. Most commonly, such Sparse Matrices are used to represent graphs or networks or to solve differential equations. Due to their high number of zero values, it is more memory-efficient to keep a dedicated array for the pointers to the non-zero values, instead of storing these matrices in their entirety. A great selection of matrix encoding formats exist [6] which are usually tweaked to exploit some property of a sparsity pattern or even to correspond with the FPGA architecture [5].

In this work, we use CSR (Compressed Sparse Row), a generic and widely used format depicted in Fig. 1. The advantage of CSR is that once the vector has been stored in cache, the matrix data can be streamed in while the result vector is streamed out. This enables a hardware design which is simple to generate with HLS. Furthermore, there is no risk of write conflicts to the result vector since a Processing Element processes an entire row at a time.

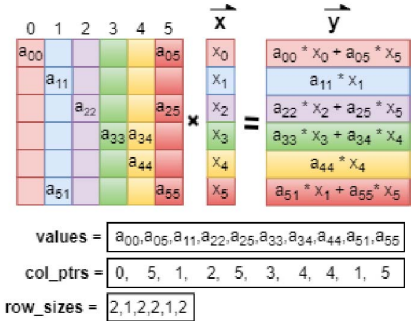

Fig. 1: SpMV with the CSR encoding.

Another common and generic encoding is the CSC (Compressed Sparse Column). The fundamental difference from CSR is that it maintains a list of the row index of the non-zero value, instead of the column index. This means that in a SpMV in the CSC format, the intermediate result vector is randomly accessed instead of $\vec{x}$.

The lack of data reuse of the matrix data, i.e. low temporal locality, is the reason the kernel is memory-bound. On the other hand, it means that only $\vec{x}$ needs to be stored in cache, while the matrix data can be streamed into the accelerator. However, it becomes difficult in large-scale problems to efficiently manage the $\vec{x}$ in cache due to each row of the matrix having non-zero values in random columns, and thus requiring random accesses to $\vec{x}$. This motivates partitioning the matrix in such a way that the range of the random accesses to $\vec{x}$ does not result in cache misses.

The most straight-forward partitioning scheme is to split the columns of the matrix vertically, thereby splitting $\vec{x}$ accordingly as shown in Fig. 2. However, this scheme produces *intermediate result vectors* $\vec{y_p}$, each with the `N` number of values, which need to be added together in order to obtain the final result vector $\vec{y}$. The more partitions created this way, the more intermediate vectors need to be added together. The other option is to split the matrix horizontally by partitioning it row-wise. However, each partition then requires the entire $\vec{x}$ in cache so this does not address the size constraint at all.

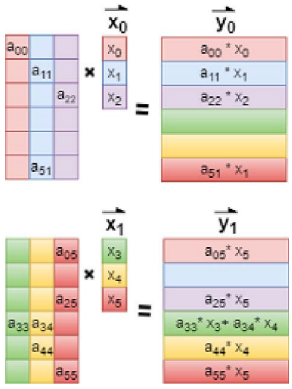

Fig. 2: The example in Fig. 1 partitioned vertically. The result vector $\vec{y}$ is obtained by adding together $\vec{y_0}$ and $\vec{y_1}$.

III. Related Works

A. Partitioning SpMV

Solutions that deal with partitioning SpMV focus on large scale problems and typically make use of column-wise partitioning in a combination with row-wise partitioning to form blocks [4], [7]. The advantage is that the additional pre-processing required is minimal, however, as we discussed previously, as many intermediate result vectors as there are partitions must be added together. In other words, the post-processing typically requires $\mathcal{O}(P \times N)$ additions where P is the number of partitions and N is the size of the result vector.

One notable solution that does not explicitly partition the SpMV is CASK [3]. A block of rows is vertically split into multiple streams which are then processed with parallel pipelines in a CSC-esque fashion. Due to this property, the size is limited by the off-chip memory and high performance is achieved as the architecture exploits parallelism within rows which has been a major performance hurdle [5]. However, due to the CSC style processing, each pipeline produces intermediate results which complicates the post-processing stage.

B. SpMV in HLS

HLS designs that map the CSR-design directly to hardware are scarce and published results achieve performance of a single MIOPS (Mega Integer Operations per Second) [8]. In

order to construct an instruction-level pipeline, it is necessary for the current version of Vivado HLS to schedule every operation in synthesis-time. In other words, the scheduling of operation is *static* and it poses a problem because each row contains a random number of non-zeros.

There is plenty of research devoted into making HLS scheduling dynamic [9], [10]. In [11] the focus is on SpMV but the performance is still far from the GFLOPS [3], [5], [12] achieved by architectural solutions, achieving only between 10 and 24 MFLOPS in simulation. The second issue of transforming the SpMV algorithm directly to hardware is to exploit the parallelism exhibited within each row.

IV. The Sparstition Algorithm

We propose the *sparstition* algorithm which splits SpMV into partitions and subsequently compresses each one in order to fit the corresponding vector in cache. This is fundamentally a row-wise partitioning scheme with the additional step of shuffling columns in the order in which they appear. The novelty of this algorithm is that it avoids the production of intermediate results. A high-level perspective of the algorithm is illustrated in Fig. 3.

The first step is to form the A_p partitions by grouping together adjacent rows via recursively partitioning the matrix into halves of equal number of rows (± 1 depending if the number of rows is odd). This may naturally lead to an unbalanced number of non-zeros across partitions for which sophisticated solutions based on graph theory exist [13], [14]. As explained in Sec. V, each row has effectively the same number of non-zeros as the hardware may insert "NOPs" into the pipeline to suit the static scheduling of Vivado HLS. Therefore the imbalance in number of non-zeros is not a concern with our design.

The second step is to compress the resulting A_p partitions to produce A'_p partitions, where the apostrophe is used to denote the post-sparstition partitions. The compression is achieved by means of shuffling the columns in the order they appear in the `col_ptrs` array from Fig. 1. If a column index is not present in a particular A_p, then the column is not created in A'_p. We must also take care of shuffling $\vec{x}$ accordingly each time we perform SpMV in iterative algorithms.

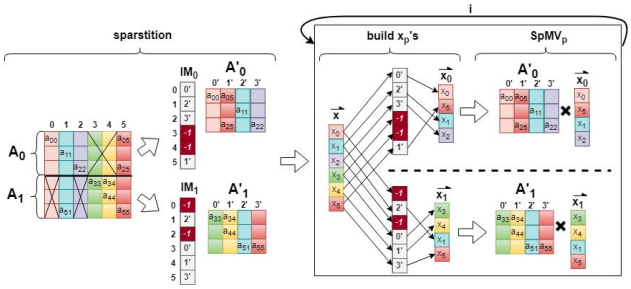

Fig. 3: The IMs are used to build the smaller $\vec{x_p}$ which are multiplied with the corresponding A'_p. The latter two steps are performed multiple times in iterative algorithms.

It is undesirable to perform the compression step multiple times especially since SpMV appears in a variety of iterative algorithms [7]. We therefore cache the mappings from A_p indices to A'_p indices with an `Index Map` (IM). Each IM has an entry for every A_p column index and is consequently of size *N*. If a column is empty for a particular A_p, such as column 3 or 4 in A_0, then the corresponding entry is populated with -1.

Listing 1 is a representation of the algorithm in pseudo-code.

Listing 1: Pseudo-code of the column shuffling part of *sparstition*.

```
/*
*invoke: sparstition(
*  nnzp, &col_ptrs[pre_cols], index_map)
*where pre_cols is the first column pointer
*  belonging to partition, and index_map has
*  been initialized.
*input: nnzp - number of non-zeros in the
*     partition
*col_ptrs: the array of column pointers
*            (from CSR)
*returns: shuffled col_ptrs array and
*           populated index_map
*/
void sparstition(int nnzp, int* col_ptrs,
    int* index_map){
   for(int nz = 0; nz < nnzp; nz++){
      int index = col_ptrs[nz];
      if(!index_map[index] == -1){
//size of x_p is the no. of non-empty columns.
         index' = x_p_size++;
         index_map[index] = index';
      } else {
        index' = index_map[index];
      }
      //shuffle the columns:
      col_ptrs[nz] = index';
   }
}
```

The pseudo-code above represents the computationally heaviest component of the *sparstition* so the complexity is roughly $\mathcal{O}(NNZ)$, where NNZ is the amount of non-zero values in the sparse matrix. The function is applied to each partition and since there is no dependence between them, speedup may be achieved by applying *sparstition* to all of them concurrently.

An optimization technique is to compress the IMs as empty columns tend to appear in groups. In the case of banded matrices, such as `epb1`, each IM will consist of close to N/N_P consecutive -1 entries, where N is the number of columns and N_P is the number of partitions. In order to save memory and to speed up the building of $\vec{x_p}$, we count the number of adjacent -1's in the new compressed IM (CIM), and replace empty entries with the negative number of the resulting count as shown in Fig. 4.

There are three configurations of the algorithm currently available, each of which has different parameters to satisfy.

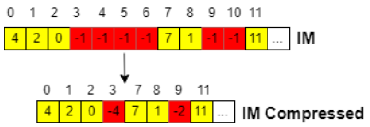

Fig. 4: IM Compression

a) the number of partitions (N_P): The algorithm divides the sparse matrix evenly into groups of adjacent rows and performs *sparstition* on each group.

b) The cache size of target accelerator: The sizes of $\vec{x_p}$ become apparent in the column-shuffling of *sparstition* so N_P is dynamically determined.

c) Both constraints: First N_P partitions are computed, then if some are too large, either only those are partitioned, or all of them (for load-balancing the number of rows).

Only the first configuration is within the scope of this work, while the others are subjects for future works.

An interesting characteristic of *sparstition* is that the output is entirely *implementation agnostic*, that is each partition is a stream which computes a segment of the SpMV product. It is therefore theoretically possible to compute the product with a variety of FPGA architectures or even GPUs. Fig. 5 illustrates this concept.

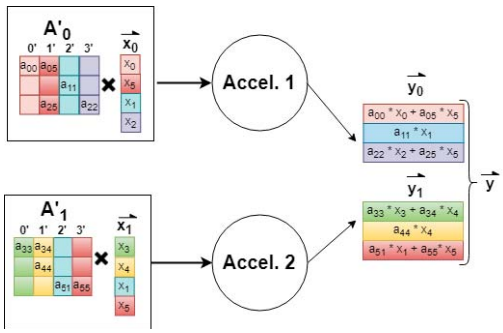

Fig. 5: The SpMV partitions can be computed using different types of accelerators.

Consequently, *sparstition* works above any existing SpMV architecture and is capable of splitting the problem and distributing it. The question now arises, whether speedup can be achieved by parallel processing once the pre-processing has been factored in.

Limitations: The *sparstition* algorithm works optimally if the A_p contains a low number of non-empty columns. Therefore the entire algorithm may run inefficiently for a matrix that contains a *dense row*. Another limitation is in the occurrence of a column index across multiple A_p partitions, which results in the replication of the corresponding $\vec{x}$. However, the replication may be minimized by carefully choosing which rows are grouped together, but that is beyond the scope of this work.

Matrices that are not impacted by these limitations, and are thus currently most suitable for the algorithm, are banded matrices commonly found in e.g. Computational Fluid Dynamics.

V. HLS Architecture

The architecture is developed on a ZedBoard which contains a ZYNQ 7000 System-on-Chip (SoC). This SoC consists of two parts, processing system (PS) and programmable logic (PL) which are connected via an interconnect. More specifically, the PS is ARM Cortex-A9 microprocessor running at 667 MHz and the PL is a Xilinx Artix-7 FPGA.

The PS and PL are interfaced via an interconnect that consists of various types of ports. The ones to our interest are the four High-Performance (HP) ports, configurable to deliver either 32- or 64-bits per cycle. At 100MHz, this amounts to 400MB/s and 800MB/s respectively. The resulting theoretical bandwidth supplied to the kernel's pipeline is 2GB/s, viz. the combination of rows, values and column streams as seen in the design in Fig. 6.

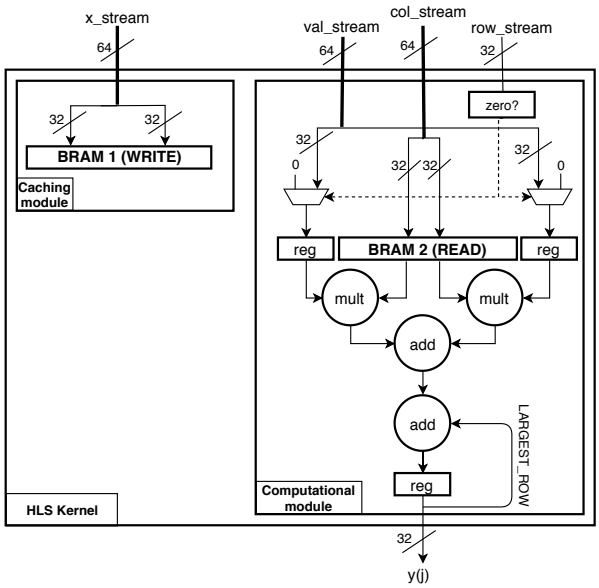

Fig. 6: A high level representation of the RTL generated, including the two modules. The two BRAMs make up the ping-pong buffer and each stores an $\vec{x_p}$.

The RTL is divided into two modules and includes a ping-pong buffer in order to pipeline caching with computation. The two modules access opposite halves of the ping-pong buffer at any given time, i.e. when one half is being written to, the other one is being read from for the computation. The design uses T2P (True Dual-Ports) BRAMs and is therefore capable of reading/writing two SP-floats each cycle with no risk of a read conflict [15].

The computation module is a simple binary tree with two leaves. Due to the static scheduling of operations in Vivado HLS, the number of products that make up the sum must be defined during synthesis-time, referred to as `LARGEST_ROW`. The most straight-forward approach, and the one opted for in the implementation, is to read from the stream until the row is completed. If the row is smaller than `LARGEST_ROW`, then the reading ceases for `LARGEST_ROW-ROW_SIZE` cycles and

zeros, which act as NOP (No Operation), are inserted into the pipeline instead. The largest row is found during pre-processing, and this number is hard-coded into the design. This implies that a collection of bitstreams may need to be generated for multiple benchmarks.

Another limitation is that the bandwidth must be a multiple of the number of leaves (i.e. 2), and is solved by zero-padding each row to have an even number of non-zeroes. This issue has been addressed in manual hardware design [16], but corresponding attempts in HLS made by the authors, where a cycle delivered part of two rows, resulted in more than a tripled clock period.

Each stream is delivered through a port, which is connected to a dedicated DMA. This architecture resembles the one presented in [17].

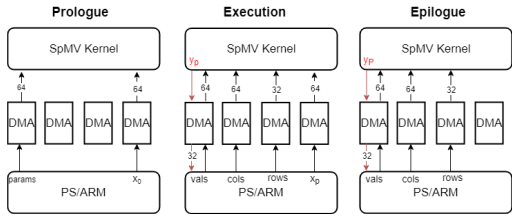

Fig. 7: High level block design as a 3-stage process. The DMA channels are configured to either 32 or 64 bits.

The computation is divided into the three following stages, which are illustrated in Fig. 7.

- **Prologue**: The initial $\vec{x_p}$ is streamed and cached in one half of the ping-pong buffer. Concurrently, two parameters are streamed as lists, which are the number of rows in each partition and the $\vec{x_p}$ sizes.
- **Execution**: The matrix data is streamed to the kernel and the result is computed with the $\vec{x_p}$ from the previous iteration.
- **Epilogue**: The last $\vec{x_p}$ is cached so traffic through this port ceases. The computation continues for one more iteration and then the $\vec{y}$ is complete.

VI. Experimental Results

A. Experimental Setup

The workstation runs with an Intel I7-8550U processor with a base frequency of 1.8GHz and 16GBs of RAM. The ZedBoard encases the ZYNQ-7020 chip that is made up of an ARM Cortex-A9 microprocessor that has the frequency 667MHz and Xilinx Series 7 FPGA fabric, which is configured to 100MHz for our experiments. The timing includes the transfer of data between ARM and the kernel, building of the $\vec{x_p}$ segments, the SpMV computation and finally *sparstition* including all additional costs such as zero-padding and the compression of `Index Maps` (IM). It excludes the pre-processing time of the matrix to CSR format, and the data transfer from the workstation to the ZedBoard.

The benchmarks are mostly obtained from the SuiteSparse Matrix Collection [18] and are found in a wide variety of domains. The choice of benchmarks used to evaluate the *sparstition* algorithm was based on the sparsity pattern of the matrix. For these reasons, there is unfortunately not much overlap with the benchmark collection commonly used. The following formula is used to compute the performance in FLOPS: $2 \times NNZ/t$, where t is the time in seconds. Matrices marked with an asterisk (*) are too large for the vector to fit in cache. The last column N_{I^*} refers to iterations until convergence with the Bi-CGSTAB solver and ILU(0) pre-conditioner. The solver solved for $A\vec{x} = \vec{b}$ where A is the benchmark matrix, $\vec{b}$ is a vector of ones, and the initial guess for $\vec{x}$ is the zero vector. The solver was considered converged when the norm of the residue was less then 10^{-8}. The largest benchmark, Hamrle3, contains zero-values on the diagonal that make them ineligible for the chosen pre-conditioner.

TABLE I: Benchmarks used to verify the *sparstition* algorithm and the HLS kernel.

Matrix	N	NNZ	Max/Avg NNZ per Row	N_{I^*}
Hummocky	12,380	120,058	11 / 9.8	105
epb1	14,734	95,053	7 / 6.45	90
wathen100	30,401	471,601	21 / 15.51	8
dixmaanl	60,000	299,998	6 / 3.0	9
epb3*	84,617	463,625	6 / 5.48	125
NORNE*	133,293	2,776,851	57 / 20.83	172
Lin*	256,000	1,766,400	7 / 6.90	12
parabolic_fem*	525,825	3,674,625	7 / 6.99	133
Hamrle3*	1,447,360	5,514,242	6 / 3.81	N/A

B. Standalone Performance of the HLS kernel

The results presented in Tab. II are grouped into two, divided by the horizontal line between `epb3` and `NORNE`, depending on whether its $\vec{x}$ (whose size corresponds with *N*) fits in the cache of the ZYNQ FPGA. The performance for the group below the line is estimated by partitioning the matrix using *sparstition* to the largest possible size so that each partition fits in the cache. Then the result is computed using the pipelined design, wherein the partitions are computed one after the other. The SpMV product was computed in software on ZYNQ's ARM microprocessor and used for verification.

TABLE II: Performance and efficiency of the HLS kernel as well as speedup compared to ARM.

Matrix	Performance (SP-MFLOPS)	P_{MAX} (SP-MFLOPS)	Speedup ARM	Efficiency
Hummocky	287.09	475.48	119.64	0.60
EPB1	273.61	464.04	117.56	0.59
wathen100	266.06	484.39	109.83	0.55
dixmaanl	281.2	454.55	135.82	0.62
epb3	309.7	458.19	150.51	0.68
NORNE	143.04	488.28	54.52	0.29
Lin	338.14	466.22	142.90	0.73
parabolic_fem	344.08	466.61	148.54	0.74
Hamrle3	252.78	441.99	98.44	0.57

There is a tremendous speedup relative to the ARM microprocessor when compared to the obtained HLS performance. In order to calculate the efficiency of the kernel, we relate the performance to the theoretical maximum performance, P_{MAX}, which is obtained by using Eq. 1. This performance would be achieved if the 2 GB/s of bandwidth delivered to the computational module were fully utilized.

$$P_{MAX} = \frac{Total\ FLOP}{Total\ bytes\ transferred} \times BW = \frac{2 \times NNZ}{size(float) \times NNZ + size(int) \times (NNZ + N)} \times 2 \quad (1)$$

Since the performance also takes into account the starting of the kernel and the prologue phase, the maximum observed efficiency is at most 0.74. We then observe that the efficiency coefficient corresponds to the statistics of each benchmark as it indicates the amount of "NOP" insertions in the computational module. For example, `LIN` and `parabolic_fem` are closest to having rows of equal *NNZ* and result in the highest efficiency. On the other hand, `NORNE` exhibits the average *NNZ* per row that is the furthest from the maximum and consequently is processed least efficiently.

C. Speedup Threshold

In order to predict whether a benchmark benefits from the *sparstition* , we first present the assumptions made in the following experiments.

The first assumption is that the partition that takes **the longest to compute is the total execution time of the application**. This is because all partitions are assumed to run in parallel and the total time of the cluster therefore depends on the accelerator that finishes last. The computation time includes both the building of $\vec{x_p}$ and the SpMV execution itself.

The second assumption is that $\vec{x}$ **arrives at each accelerator at exactly the same time**. This requires the host to have high bandwidth in order to stream the vector simultaneously to a number of nodes, and becomes more unrealistic with an increasing number of partitions.

The third assumption is that there is **no overhead in the iterative algorithms to commence parallel SpMV**. We make this assumption because the overhead cost is minimized if there is efficient pipelining between the steps of the solver algorithm.

We define a new metric, under the name *Number of Iterations for Speedup* (N_I), in order to more readily determine if an iterative algorithm benefits from the *sparstition* for the chosen benchmarks. This number tells us how many iterations it takes for the pre-processing "debt" to be paid up, given that each partition is computed in parallel. It is computed as follows:

$$T_{Sp} \times N_I = T_S + T_P \times N_I$$
$$N_I = \frac{T_s}{T_{Sp} - T_P} \quad (2)$$

The variables of Eq. 2 and their relationships are illustrated in Fig. 8. There are two possibilities we explore once the data has been loaded into memory and the SpMV product is ready to be computed. The first possibility is to compute the SpMV without any partitioning, which will take time T_{Sp}. The other one performs the *sparstition* algorithm at the time-cost T_S and computes the partitioned SpMV in time T_P. T_P includes the

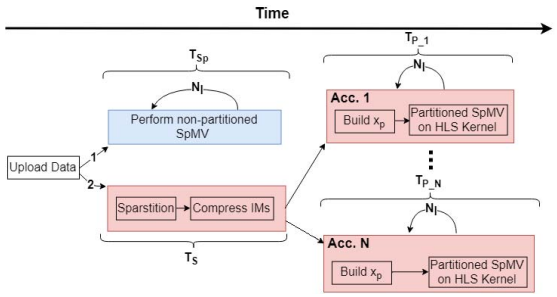

Fig. 8: Illustration of the computation on whether *sparstition* is beneficial.

building of x_p and the partitioned SpMV, and is taken to be the longest execution time of a partition.

For the following calculations, the time it takes to read the matrix file and generate the CSR-format is excluded from the pre-processing cost. All other steps, such as zero-padding and *sparstition* are included.

Fig. 9 illustrates the N_I for every benchmark. There is more speedup to be gained moving from 2 partitions to 4 partitions, but the speedup levels out thereafter and then the N_I increases linearly.

In order to explain this eventual rise, the *sparstition* algorithm can be broken into two significant stages when considering the computational cost: the initialization and the actual partitioning. The initialization stage consists of initializing the IMs in order to keep track of whether a mapping is new. Each partition requires an IM, thus this stage is dependent on the number of partitions (N_P). The next stage is the functional aspect of the algorithm. This stage is independent of N_P, because as N_P grows, each partition in turn becomes smaller.

Therefore, as N_P rises, the more IMs need to be initialized which causes the rise in N_I. This effect would be alleviated by parallelising the initialization stage, which is not currently exploited. Furthermore, it is unreasonable to expect there would be more than 2^6 accelerators available.

Fig. 9: The number of iterations until speedup is achieved with the assumption that the partitions are computed in parallel.

From the rightmost column in Table I we see that N_{I^*} is at

least 8 and even exceeds 100 iterations for almost half of the benchmarks. Since N_I is 3 or less for every benchmark with 2^6 partitions, speedup is expected for every benchmark. The expected speedup is formulated as follows.

$$speedup = \frac{T_{Sp} \times N_{I^*}}{T_S + T_P \times N_{I^*}} \quad (3)$$

The numerator of Eq. 3 reflects the total time of running SpMV unpartitioned, and the denominator that when *sparstition* is applied. Notice that if it takes less time to compute SpMV without *sparstition* then $speedup < 1$, as expected. From Fig. 10 we deduce that the speedup is first and foremost dependent on how large N_{I^*} is, but also the size of the benchmark and efficiency.

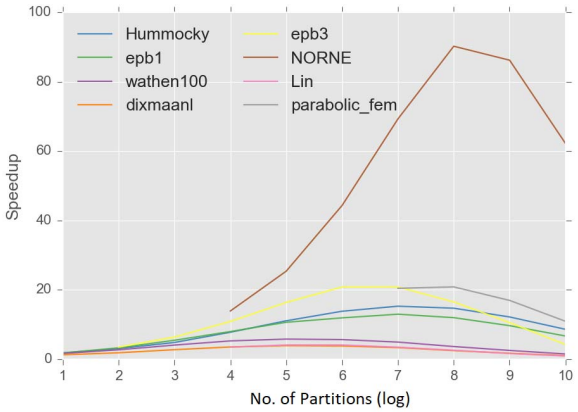

Fig. 10: The expected speedup (unpartitioned vs. sparstitioned) when performing N_{I^*} iterations of SpMV

Each benchmark contains a sweet-spot when the overhead of initializing an increasing number of IMs outweighs the speedup gained from further paralellising. However, the parallelism in initializing the IMs is currently not being exploited, and therefore a linear increase in speedup is feasible.

D. Performance of Sparstitioned SpMV

Tab. IV summarizes the results from executing all the partitions on the HLS kernel. In the left-most column is the time it took for the non-partitioned SpMV product to compute. We otherwise see that T_S predictably rises as the N_P increases, while T_P decreases. Notice that the lowest possible T_S is 0.03ms, but at this point each partition is just a couple of rows. The time measured in these cases is the communication overhead with the kernel.

Fig. 11 shows the estimated performance up to 64 partitions when using High Bandwidth Memories (HBM).

The predicted results when multiple kernels are executing in parallel exceeds 10 GFLOPS which is unprecedented for simple SpMV kernels in HLS.

E. Comparison with State-of-the-Art

The comparison is split in two parts, one that evaluates the performance to the state-of-the-art architectures developed

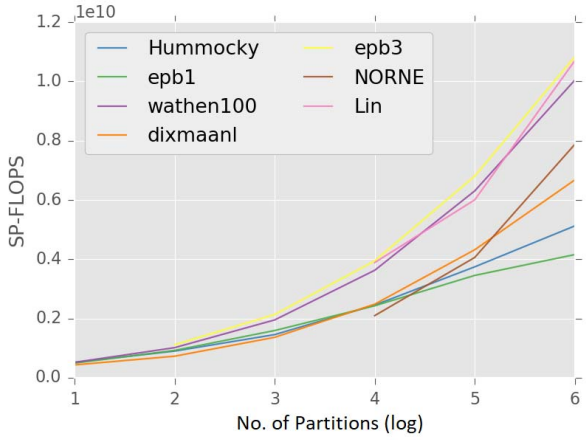

Fig. 11: The estimated performance using T_P of benchmarks up to 2^6 partitions with HBM, assuming that each kernel is supplied with a bandwidth of 2GB/s.

with an HDL (Hardware-Description Language) and the advanced architectural Dataflow language solution CASK. The other part compares against the best performing HLS compiler solutions.

1) HDL and Advanced HLS Solution: When comparing our HLS kernel to the current state-of-the-art [3], [12], [19], it quickly becomes clear that there is still a long way to go. The HLS kernel achieves performance at around 300 MFLOPS using single-precision with a bandwidth of 2GB/s. In order to compete with the best results, it should be able to achieve double-precision performance in the range of 2-14 GFLOPS.

The relatively low bandwidth of the ZYNQ system is certainly a factor. Once *sparstition* comes into play and given that it is distributed efficiently to the computing elements of a cluster-like systems, then multiple instances of the HLS kernel may boost the performance considerably.

2) HLS Compiler: There is not a lot of work done on SpMV kernels developed with HLS. [8] presents a HLS survey with SpMV as a case study, but the algorithm is synthesized directly without any attempt to achieve any (near-)optimal implementation with good performance. The presented result only achieves a couple of MIOPS.

TABLE III: Comparison of this work with [11] for their benchmarks in double precision.

Matrix		Execution Time (ms)			
Name	Max NNZ per Row	[11]	This work	Bandwidth (GB/s)	Speedup
bcsstm25	6	2.8	2.15	0.29	1.3
dw8192	8	3.5	0.77	1.02	4.6
bcsstk12	27	1.0	0.46	0.68	2.2
ex7	75	3.0	1.29	0.70	2.3
poli3	336	5.5	N/A	N/A	N/A

The current state-of-the-art SpMV work that used an HLS compiler to directly map the algorithm to hardware is presented in [11]. To achieve an accurate comparison, we will configure the kernel to compute in double-precision which

TABLE IV: Table summarizing the *sparstition* cost (T_S) and the execution time of the slowest partition (T_P). All times are in milliseconds.

Benchmark \ No. of Partitions	1	2		4		8		16		32		64		128		256		512		1024	
	T_{Sp}	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P	T_S	T_P
Hummocky	0.84	0.90	0.46	0.95	0.27	1.03	0.17	1.07	0.10	1.21	0.06	1.44	0.05	1.77	0.04	2.45	0.03	3.85	0.03	6.95	0.03
epb1	0.69	0.66	0.39	0.68	0.21	0.71	0.12	0.78	0.08	0.90	0.06	1.11	0.05	1.48	0.04	2.10	0.03	3.44	0.03	6.48	0.03
wathen100	3.54	3.11	1.82	3.19	0.93	3.15	0.48	3.33	0.26	3.72	0.15	4.29	0.09	5.22	0.07	7.39	0.06	11.05	0.05	18.97	0.05
dixmaanl	2.13	3.11	1.39	3.19	0.83	3.15	0.44	3.33	0.24	3.72	0.14	4.29	0.09	5.22	0.07	7.39	0.06	11.05	0.05	18.97	0.05
epb3	2.99			4.40	0.83	4.63	0.43	5.02	0.24	5.85	0.14	7.27	0.09	10.25	0.06	16.51	0.05	29.71	0.05	83.49	0.04
NORNE	38.83							25.87	2.65	27.92	1.37	29.36	0.71	32.86	0.37	38.87	0.20	54.62	0.13	90.70	0.10
Lin	10.45							25.41	0.91	23.80	0.59	26.92	0.33	34.50	0.18	47.95	0.13	78.72	0.06	141.54	0.05
parabolic_fem	21.36													73.74	0.49	98.91	0.28	146.54	0.16	250.00	0.08
Hamrle3	43.80																	495.94	0.33	900.71	0.24

implies that the ports specified in the design in Fig. 6 deliver a single value per cycle. The results are presented in Tab. III.

It is important to note that the results in [11] are obtained from simulation. It was unfortunately infeasible for us to synthesize a pipeline large enough to compute the largest row of `poli3` due to the enormous amount of routing and scheduling required. For all other benchmarks, however, speedup is achieved.

The solution in [11] may be more efficient as it aims to mitigate the need for the aggressive "NOP" insertion of our design. However ours uses as much bandwidth as possible which is critical for memory-bound applications. The two solutions would work well together.

VII. Conclusion & Future Work

We introduced in this paper the *sparstition* algorithm, which enables the parallel processing of SpMV without the production of intermediate results. We tested our solution on ZYNQ, where matrices as large as $9\times$ the available cache were computed. A corresponding architecture developed with Vivado HLS, which pipelines and overlaps memory transfer and computation was also presented. The results obtained from our kernel outperforms other similar solutions as we exploited as much bandwidth as possible, a critical aspect when dealing with memory-bound functions. We also presented our performance estimations when using High Bandwidth Memory, which would boost the performance to GFLOPS.

Future Work: There is work planned to extend the *sparstition* algorithm in the future. 1) Currently, *sparstition* is being integrated with a manually implemented iterative solver. This may enable the solving of matrices much larger than the available cache of the accelerator. 2) Validate the work in this paper with High Bandwidth Memories (HBM) (when available to us) to compute the partitions in parallel. 3) To demonstrate the implementation-agnostic attribute and to demonstrate speedup in iterative algorithms when using HBMs.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120.

[2] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Oct 2015.

[3] P. Grigoras, P. Burovskiy, and W. Luk, "Cask: Open-source custom architectures for sparse kernels," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 179–184.

[4] F. Sadi, L. Fileggi, and F. Franchetti, "Algorithm and hardware co-optimized solution for large spmv problems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017.

[5] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014.

[6] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003, aAI3121741.

[7] J. Peng, Z. Xiao, C. Chen, and W. Yang, "Iterative sparse matrix-vector multiplication on in-memory cluster computing accelerated by gpus for big data," in *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, Aug 2016, pp. 1454–1460.

[8] S. Skalicky, C. Wood, M. ukowiak, and M. Ryan, "High level synthesis: Where are we? a case study on matrix multiplication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013.

[9] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 127–136.

[10] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 78–85.

[11] R. Garibotti, B. Reagen, Y. S. Shao, G. Wei, and D. Brooks, "Assisting high-level synthesis improve spmv benchmark through dynamic dependence analysis," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1440–1444, Oct 2018.

[12] J. Zambreno and K. Townsend, "Reduce, reuse, recycle (r3): A design methodology for sparse matrix vector multiplication on reconfigurable platforms," in *Proceedings of the 2013 IEEE 24th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, ser. ASAP '13, 2013.

[13] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2d graph partitioning," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2013, pp. 1–12.

[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[15] "7 Series FPGAs Memory Resources, User Guide (UG473)."

[16] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, ser. FPGA '05. New York, NY, USA: ACM, 2005.

[17] J. Pinhao, W. Jose, H. Neto, and M. Vestias, "Sparse matrix multiplication on a reconfigurable many-core architecture," in *2015 Euromicro Conference on Digital System Design*, Aug 2015, pp. 330–336.

[18] The suitesparse matrix collection. [Online]. Available: https://sparse.tamu.edu/

[19] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13, 2013, pp. 3:1–3:10.